HP64000 Logic Development System

Model 64815AF Pascal/64000 Compiler Supplement 68000/68008/68010

HEWLETT PACKARD

SOFTWARE NOTICE Manual Part No. 64815-90906

NOTICE

Attached to this software notice is a summary of problems and solutions for the 68000 Pascal compiler that you may or may not encounter. Use this summary with the manual you received with the product. In the one-line description at the top of each problem and solution, there is a software topic or manual chapter reference.

KPR #: 5000096313 Product: 68000 PASCAL M64815-90906 01.00

Keywords: RUN-TIME LIBRARY

One-line description:
Zenter_trap is not compatible with 68010. (See Chap 3, pg 3-5)

Problem:
When the compiler generates code for a trap procedure with parameter passing, it generates a call to Zenter_trap. This works for the 68000, but not the 68010.

Solution:
Create a subroutine with the same name, declare it global, and link it before the libraries. This way the user defined routine will be used instead of the one in the library.

KPR #: 1650012427 Product: 68000 PASCAL M64815-90906 01.09

Keywords: MANUAL

One-line description:
With mult on real var libraries gen privileged instruction for 68010.

Problem:
When doing multiplication on real variables, the 68000/08/10 compiler libraries generate a privileged instruction for the 68010 microprocessor, which can only run in supervisor mode. The following instruction needs to be changed in the Zreal_mul library.

```
        MOVE     SR,D3
```

Solution:
Change the above instruction to:

```
             MOVE    CCR,D3
```

You can obtain a copy of the library sources from an HP field office.

KPR #: D200036897   Product: 68000 PASCAL   M64815-90906   01.09

Keywords: TYPE CONVERSION

One-line description:
Signed_8 to Unsigned_16 is incorrect.

Problem:

```
VAR S8 : SIGNED_8;
    US16 : UNSIGNED_16;

BEGIN
US16 := UNSIGNED_16(S8);  This does a sign extend which is incorrect.
```

Solution:
None at this time.

FOLD HERE

HEWLETT
PACKARD

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY CARD
FIRST CLASS PERMIT NO. 1303 COLORADO SPRINGS, COLORADO

POSTAGE WILL BE PAID BY ADDRESSEE

HEWLETT-PACKARD
Logic Product Support Dept.
Attn: Technical Publications Manager
Centennial Annex - D2
P.O. Box 617
Colorado Springs, Colorado 80901-0617

FOLD HERE

Your cooperation in completing and returning this form
will be greatly appreciated. Thank you.

READER COMMENT SHEET

Operating Manual, Model 64815AF
Pascal/64000 Compiler Supplement 68000/68008/68010
64815-90906, May 1984

Your comments are important to us. Please answer this questionaire and return it to us. Circle the number that best describes your answer in questions 1 through 7. Thank you.

1. The information in this book is complete:

Doesn't cover enough (what more do you need?)	1 2 3 4 5	Covers everything

2. The information in this book is accurate:

Too many errors	1 2 3 4 5	Exactly right

3. The information in this book is easy to find:

I can't find things I need	1 2 3 4 5	I can find info quickly

4. The Index and Table of Contents are useful:

Helpful	1 2 3 4 5	Missing or inadequate

5. What about the "how-to" procedures and examples:

No help	1 2 3 4 5	Very helpful
Too many now	1 2 3 4 5	I'd like more

6. What about the writing style:

Confusing	1 2 3 4 5	Clear

7. What about organization of the book:

Poor order	1 2 3 4 5	Good order

8. What about the size of the book:

too big/small	1 2 3 4 5	Right size

Comments: ____________________

____

____

____

Particular pages with errors?

____

Name (optional): ____________________
Job title: ____________________
Company: ____________________
Address: ____________________

Note: If mailed outside U.S.A., place card in envelope. Use address shown on other side of this card.

FOLD HERE

HEWLETT
PACKARD

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY CARD

FIRST CLASS PERMIT NO. 1303 COLORADO SPRINGS, COLORADO

POSTAGE WILL BE PAID BY ADDRESSEE

HEWLETT-PACKARD
Logic Product Support Dept.
Attn: Technical Publications Manager
Centennial Annex - D2
P.O. Box 617
Colorado Springs, Colorado 80901-0617

FOLD HERE

Your cooperation in completing and returning this form
will be greatly appreciated. Thank you.

READER COMMENT SHEET

Operating Manual, Model 64815AF
Pascal/64000 Compiler Supplement 68000/68008/68010
64815-90906, May 1984

Your comments are important to us. Please answer this questionaire and return it to us. Circle the number that best describes your answer in questions 1 through 7. Thank you.

1. The information in this book is complete:

Doesn't cover enough (what more do you need?)	1 2 3 4 5	Covers everything

2. The information in this book is accurate:

Too many errors	1 2 3 4 5	Exactly right

3. The information in this book is easy to find:

I can't find things I need	1 2 3 4 5	I can find info quickly

4. The Index and Table of Contents are useful:

Helpful	1 2 3 4 5	Missing or inadequate

5. What about the "how-to" procedures and examples:

No help	1 2 3 4 5	Very helpful
Too many now	1 2 3 4 5	I'd like more

6. What about the writing style:

Confusing	1 2 3 4 5	Clear

7. What about organization of the book:

Poor order	1 2 3 4 5	Good order

8. What about the size of the book:

too big/small	1 2 3 4 5	Right size

Comments: ____________________

____

____

____

Particular pages with errors?

____

Name (optional): ____________________
Job title: ____________________
Company: ____________________
Address: ____________________

Note: If mailed outside U.S.A., place card in envelope. Use address shown on other side of this card.

Pascal/64000
Compiler Supplement
68000/68008/68010

Printing History

Each new edition of this manual incorporates all material updated since the previous edition. Manual change sheets are issued between editions, allowing you to correct or insert information in the current edition.

The part number on the back cover changes only when each new edition is published. Minor corrections or additions may be made as the manual is reprinted between editions. Vertical bars in a page margin indicate the location of reprint corrections.

First Printing ..... November 1981 (P/N 64815-90902)
Second Edition .... June 1982 (P/N 64815-90903)
Third Edition .... August 1982 (p/N 64815-90904)
Fourth Edition ..... February 1984 (P/N 64815-90905)
Fifth Edition .... May 1984 (P/N 64815-90906)

Table of Contents

Table of Contents (Cont'd)

List of Tables

Chapter 1

PASCAL/64000 COMPILER

INTRODUCTION

This compiler supplement is an extension of the Pascal/64000 Compiler Reference Manual. It contains processor-dependent compiler information for use with 68000, 68008, and 68010 microprocessors.

NOTE

All references to the 68000 microprocessor in this manual are equally applicable to the 68010 microprocessor unless otherwise noted.

Descriptions of compiler features, options, and their uses are supplied. A discussion of run-time libraries required by the 68000 code generator is included. In addition, a brief discussion of the features, capabilities, and limitations of Pascal program development using the emulator is provided.

PASCAL PROGRAM DESIGN

Pascal programs should be designed to be as processor- and implementation-independent as possible, yet certain concessions must be made when the processor has unique characteristics. For example, most large-mainframe computer implementations have enough memory to allocate a stack area and a heap for dynamic memory allocation with no prompting by the user. For the 68000, the user must specify the location of the stack, and if needed, the location of a memory pool for dynamic allocation routines. The following sections describe subjects related to programming and compiling Pascal/64000 programs for the 68000 processor.

HOW TO IMPLEMENT A PROGRAM

The usual process of software generation is as follows:

a. Create source program files using the editor.
b. Compile source programs.
c. Link the relocatable files.
d. Emulate the absolute file.
e. Debug as necessary.

This chapter will provide insight into each of these processes.

THE SOURCE FILE

The Pascal/64000 compiler takes as input a program source file created with the editor. The basic form of a source file is:

```
"68000"
PROGRAM Name;
  .               {comments}
  .
CONST
    ...;
    ...;
TYPE
    ...;
    ...;
VAR
    ...;
    ...;
PROCEDURE Procedure_name(Parameter1 : Type);
    .
    .
    BEGIN
      .
      .
      .
    END;
BEGIN
  .
  .               {main program code}
  .
END.
```

When source file editing is complete, the file is ready for compilation. Notice in the example form that the first line of the source program specifies the name of the processor. This first line must always be present.

Within a Pascal source program, the compiler recognizes only upper-case keywords, but identifiers may be lower case. There are five possible forms of compiler output: a relocatable file, a listing file (if specified), an assembly language source file (if specified), an assembly symbol file, and a compiler symbol file, and a compiler symbol file. These output files are described in the following paragraphs.

Relocatable file:	If no errors were detected in the source file (called FILENAME:source), a relocatable file (called FILENAME:reloc) will be created. This file will be used by the linker to create an executable absolute file.
Listing file:	If a listfile is specified, a listing file containing source lines with line numbers, program counter, level numbers, errors and expanded code (if specified) will be generated.
Assembly file:	If the ASM_FILE option is turned on anywhere in the source file, an assembly file (called ASM68000:source) will be created. This file will contain the Pascal source as comments and the assembly language produced. This file may be assembled with the 68000 assembler. There is only one ASM68000 source file per userid. On multistation systems this means that when two 68000 compilations are done simultaneously n the same userid and both sources contain the ASM_FILE option, only one of the compilations will build the ASM68000 file.
Assembly symbol file:	If no errors were detected in the file (called FILENAME:source), an assembly symbol file (called FILENAME:asmb_sym) will be created. This file contains information about symbols that were defined in the source file and is used by the various HP 64000 analysis tools during the debugging stage. The :asmb_sym file may be suppressed by using the $ASMB_SYM OFF$ compiler directive.
Compiler symbol file:	If the compiler was executed using "options comp_sym", a compiler symbol file (called FILENAME:comp_sym) will be created. The file contains additional high-level information about symbols defined in the source file and is used by the HP 64000 high-level analysis tools during the debugging stage.

PRODUCING PROGRAMS FOR THE 68008 PROCESSOR

When compiling a program for the 68008 processor, the first line of the source file must contain the special compiler directive "68008" as shown below.

```
"68008"
PROGRAM NAME;

....

BEGIN

....

END.
```

When linking for the 68008, all modules must be compiled or assembled using the "68008" directive or the linker will produce an error. In particular, the user must specify the 68008 run-time library file names, A5_LIB:L68008, etc., as opposed to the 68000 library files. Refer to "Linking" in the paragraphs below.

When compiling for the 68008 processor, the compiler produces exactly the same instructions as for the 68000 processor. The only difference is the relocatable files are identified as being for an 8-bit processor as opposed to a 16-bit processor. This attribute only affects the operation of the PROM Programmer.

LINKING

After all program modules have been compiled (or assembled), the modules may be linked to form an executable absolute file. The compiler generates calls to a set of library routines for commonly used operations such as input/output, signed and unsigned multiply and divide for 32-bit numbers, dynamic memory allocation, real number processing, etc. These routines must be linked with the program modules.

These routines are provided in seven libraries for the 68000 and seven other libraries for the 68008 processor. The names of the library relocatable files and a description of their functions is given in Table 1-1. The library files in userid L68000 should be linked with modules compiled for the 68000 processor while the library files in userid L68008 should be linked with modules compiled for the 68008 processor.

Note that there are two library files implementing the same functions in all cases except for the real number library. For instance, A5_LIB and ABS_LIB both implement dynamic memory allocation, 32-bit arithmetic, etc. The "A5" and "ABS" versions of a particular library define the same routines and are identical in function. They differ only in the method used for accessing their own global variables. Refer to the section on Addressing Options in Chapter 2 for more information.

Table 1-1. Relocatable Library Files

Library File Name	Global Variable Addressing Mode	Functions Provided
A5_LIB:L68000 A5_LIB:L68008	A5 relative (i.e. $COMMON$)	Dynamic memory allocation, 32-bit arithmetic, string operations, range checking, program starting and ending.
ABS_LIB:L68000 ABS_LIB:L68008	absolute (i.e. FAR)	Same as A5_LIB except for global variable addressing mode.
REAL_LIB:L68000 REAL_LIB:L68008	N/A (no global variables used)	Real number operations.
A5_PIOLIB:L68000 A5_PIOLIB:L68008	A5 relative (i.e. $COMMON$)	Pascal I/O routines to implement RESET, READ, WRITE, etc.
ABSPIOLIB:L68000 ABSPIOLIB:L68008	absolute (i.e. FAR)	Same as A5_PIOLIB except for global variable addressing mode.
A5_SIMLIB:L68000 A5_SIMLIB:L68008	A5 relative (i.e. $COMMON$)	Simulated I/O primitive routines for emulation environment.
ABSSIMLIB:L68000 ABSSIMLIB:L68008	absolute (i.e. FAR)	Same as A5_SIMLIB except for global variable addressing mode.

You may replace one or more of the routines in the libraries with your own version simply by specifying your own routines at link time.

LINKING EXAMPLE

The linker is called and the questions asked by the linker should be answered as follows:

```
link   ...

Object files: MODULE1,MODULE2,USER_LIB

Library files: A5_LIB:L68000

Load addresses: PROG,DATA,COMN,A5=0004000H,0002000H,0000000H,
                                                    0000000H
    .
    .
    .

Absolute file name: PROGRAM
```

Any object files that are meant to replace files in library files A5_LIB, ABS_LIB, REAL_LIB, A5_PIOLIB, ABSPIOLIB, A5_SIMLIB, or ABSSIMLIB should be contained in the library "USER_LIB" above.

The numbers specified for the PROG, DATA, COMN, and A5 values have the following meaning.

PROG - Specifies the starting location of the "program" memory area. Machine instructions and constant data are stored in the PROG area. The value of PROG must be an even address. In the example above, PROG is set to 4000H so the emulator will load the absolute file at 4000H. Note that, in general, the address specified in PROG is not the entry point where the emulator will begin execution of the absolute module.

DATA - Specifies the starting location of the DATA relocatable memory area. Program-level variables (i.e. any variable outside of a Pascal procedure or function) are normally the DATA area.

The value of DATA must be an even address. In the above, the linker will combine the DATA segments from each of the relocatable files above into one and load it address 2000H.

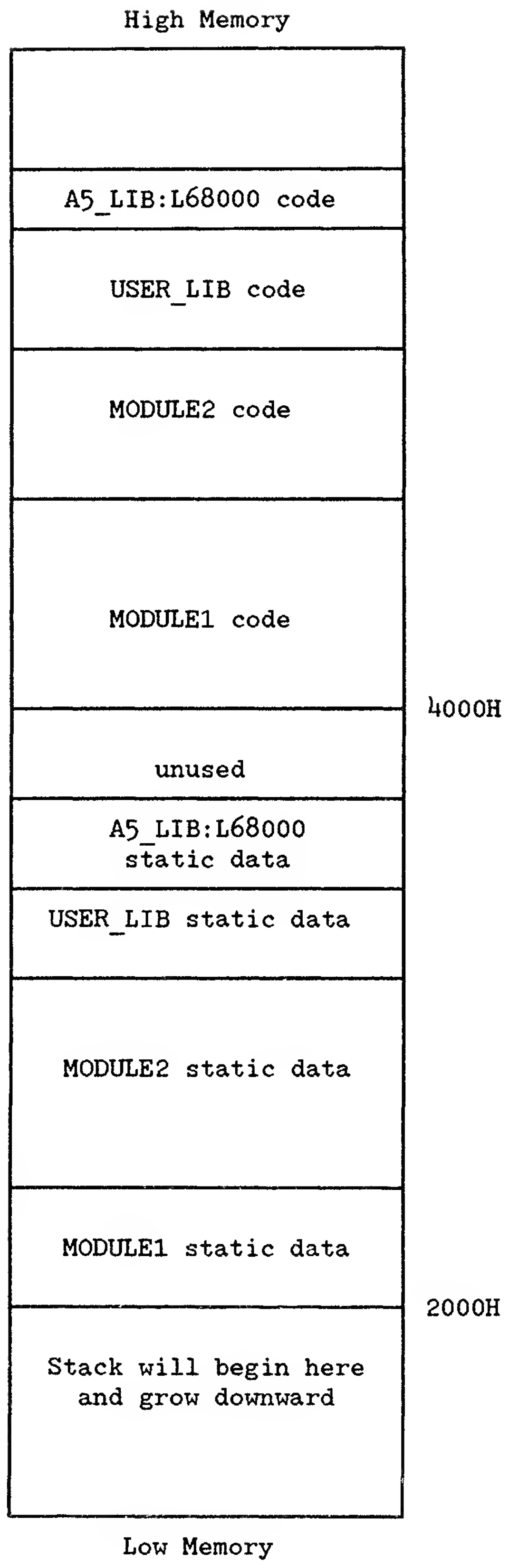
COMN - Specifies the starting location of the COMN relocatable memory area. The compiler never allocates data to the COMN area. The COMN area is seldom used. It is meant to provide for a FORTRAN-like sharable common area.

A5 - Specifies the value that will be contained in the 68000 address register A5. A5 is a register used by the compiler for accessing program level variables when the $COMMON$ compiler option is in effect. Refer to the section on program-level variable addressing which follows.

The value of A5 must be an even address. A5 should be set to the value that will actually be loaded into address register 5 at run time.

The following diagram shows how the absolute file would look in memory after linking and loading. It is assumed that the value 2000H will be loaded into register A7 to define the beginning address of the hardware-maintained user stack.

When library routines from the library A5_LIB:L68000 are required they will be linked at the end of the last user-relocatable PROG and/or DATA areas.


High Memory
A5_LIB:L68000 code
USER_LIB code
MODULE2 code
MODULE1 code
4000H
unused
A5_LIB:L68000 static data
USER_LIB static data
MODULE2 static data
MODULE1 static data
2000H
Stack will begin here and grow downward
Low Memory

PROGRAM-LEVEL VARIABLE ADDRESSING

The compiler can use any one of three addressing modes to access any particular program-level (i.e. not defined within a Pascal routine) variable. The compiler options COMMON, BASE_PAGE, and FAR are used to select the program-level variable addressing mode in the following manner.

$COMMON$	specifies that affected variables are accessed using Address Register Indirect With Displacement mode. The address register used is A5. This is the default addressing mode.
$BASE_PAGE$	specifies that affected variables are accessed using Absolute Short Address mode.
FAR	specifies that affected variables are accessed using Absolute Long Address mode.

The addressing option, COMMON, BASE_PAGE, or FAR, that is in effect when the keyword PROGRAM is encountered in a source file controls the addressing mode for all program-level variables defined within that program. After the keyword PROGRAM, the options COMMON, BASE_PAGE, and FAR only affect the addressing of external variables. Consider the following example.

```
"68000"
PROGRAM P;
VAR A:INTEGER;
$FAR$
$EXTVAR ON$
    B:INTEGER;
$EXTVAR OFF$
    C:INTEGER;
BEGIN
A:=0; { A accessed w/Address Register Indirect With Displacement mode }
B:=0; { B accessed w/Absolute Long Address mode                        }
C:=0; { C accessed w/Address Register Indirect With Displacement mode }
END.
```

In the example above, the $COMMON$ option, by default, is in effect when the keyword PROGRAM is encountered. Therefore all program-level variables defined in this program (i.e. A and C) will be accessed in the $COMMON$ mode in this program. Note that FAR, specified after the keyword PROGRAM, affects only the externally defined variable B.

The $COMMON$ option causes affected variables to be accessed using the A5+d addressing mode. In this case, d is a sixteen bit displacement which occupies two bytes of memory in the machine instruction. The displacement is first sign extended to 32 bits and then added to the contents of A5 to produce the effective address of the variable. The 16 bit displacement value allows a maximum of 64k bytes to be accessed in the DATA area. Because the 16 bit displacement is sign extended, the area of memory that can be accessed is +/- 32k bytes on either side of the address contained in A5.

The linker calculates the displacement value, d, of an A5+d variable reference in the following way. The A5 value, specified at link time, is subtracted from the actual address of the referenced variable. The result, truncated to 16 bits, becomes the displacement value. The linker will report an error if the calculated displacement is greater than 32767 or less than -32768.

The library routine Zstartprogram contained in A5_LIB or ABS_LIB sets the value of A5 to 0000000H. The linker initially sets its A5 value to 0000000H. By default then, the program may access memory locations in the range 0000000H through 0007FFFH for positive displacement values and 0FF8000H through 0FFFFFFH for negative displacement values assuming a 24 bit address width. If one desires to locate the DATA area outside of the above ranges, one must specify a new value for A5 at link time and also link to an assembly language routine which executes code to load A5 with the same new value.

The $BASE_PAGE$ option causes affected variables to be accessed using the Absolute Short Address mode. In this case, a 16 bit absolute address is contained in two bytes of memory in the machine instruction. The 16 bit address is sign extended to 32 bits to produce the effective address of the variable. The 16 bit address allows a maximum of 64k bytes to be accessed in the DATA area. Because the 16 bit address is sign extended, the accessible memory locations are in the range 0000000H through 0007FFFH for positive values and 0FF8000H through 0FFFFFFH for negative values assuming a 24 bit address width. The linker will report an error if the actual address of the referenced variable is outside these ranges.

The FAR options causes affected variables to be accessed using the Absolute Long Address mode. In this case, a 32 bit absolute address is contained in four bytes of memory in the machine instruction. The 32 bit address is the same as the effective address. This mode allows any location in memory to be accessed but at the cost of longer code and slower execution.

One may access a particular variable using different modes from different modules. For example, one module may define a global variable with the $COMMON$ option in effect and access that variable using A5+d mode within that module. Another module may declare the same variable external with the $BASE_PAGE$ or FAR option in effect and access the variable using Absolute Short or Absolute Long mode. Of course, when using $COMMON$ or $BASE_PAGE$, the actual address of the variable must always be within the range that is accessible using A5+d or absolute short addressing modes respectively.

POSITION INDEPENDENT CODE AND DATA

Several compiler options are available to the user to help him control the 68000 addressing modes used by the compiler when it generates code. The options are described in Chapter 2 under the heading Addressing Options. With these options it is possible for the user to postpone the determination of the load addresses for code and data until run time. This can be useful in a multiprogramming environment where dynamic memory mapping is required but the hardware to do it is not available.

If either the CALL_PC_SHORT or the CALL_PC_LONG option is used and either the LIB_PC_SHORT or the LIB_PC_LONG option is used, the executable code generated by the compiler will be position independent, and may be loaded anywhere in memory at run time. However, the linker requires that an address be specified for PROG, and the emulation loader will always load the absolute file generated by the linker at that address for emulation purposes.

If the $COMMON$ option is used before the keyword PROGRAM, all references to program level variables (variables in the static data block) will be accessed using the Address Register Indirect Plus Displacement addressing mode. The address register used will be A5. The executable code generated by the compiler will be position independent and may be loaded anywhere in memory at run-time. Also, the data relocatable area may be located anywhere in memory at run time. However, the linker requires that addresses be specified for DATA and A5. The emulation loader will always load the data portion of the absolute file at the location specified for DATA.

If the user loads the data to a different location, then, for correct execution, the user must insure that difference, DATA - A5, specified at link time is equal to the difference of the actual run time location of the data area minus the actual run time value of A5.

The COMMON option is on by default at compiler initialization time. To cause the compiler to use absolute addressing for program level variables use the BASE_PAGE option or the FAR option before the keyword PROGRAM.

EMULATION OF PASCAL PROGRAMS

After all modules have been compiled (or assembled) and linked, the absolute file may be executed using the emulation facilities of the Model 64000. The emulator should be configured with the memory mapped as it will be used in the target system.

A program which is designed to run in read-only-memory (ROM) may be compiled with the 68000 compiler. The $SEPARATE$ option described in the 64000/Pascal Reference Manual is ignored by the 68000 code generator. Pascal programs compiled for the 68000 are always compiled as if the $SEPARATE$ option is on. Thus, RAM data will always be counted under the DATA counter, while code and constants are always counted under the PROG counter.

In the linker example above, either MODULE1 or MODULE2 has a program block. That is, one of the modules has:

```
BEGIN

  Pascal statements

END.
```

and the other module has a period with no BEGIN-END block after the procedure declarations. The BEGIN in the module with the BEGIN-END block will be labeled by the compiler with the program name of that module. The linker establishes this name as the entry point of the absolute file. This program may be executed in the emulator by the command:

```
run from <program name>
```

where <program name> is the name of the program (MODULE1 or MODULE2) that has the BEGIN-END block.

If two Pascal programs are linked together and both programs have BEGIN-END blocks, the linker will indicate a multiple transfer address. The following statements are equivalent:

1. An absolute file should have zero or one specified entry points at link time.
2. When linking Pascal programs, a maximum of one program should have a BEGIN-END block.

When executing code on the emulator, if a run time error occurs, a jump to the monitor will be generated and a message will be displayed on the status line indicating the error and, in many cases, the address where the error occurred.

NOTE

It is important to remember that during emulation of Pascal/64000 programs, a Pascal program may be debugged symbolically (using global symbols in the source program) or by source program line numbers of the form: #n, where n is the source line number of an executable Pascal line.

Chapter 2

PASCAL/64000 PROGRAMMING

PROGRAMMING CONSIDERATIONS

This chapter describes the run-time environment for 68000 Pascal/64000 programs. Although some parts of the run time environment are not necessary for every Pascal program, the programmer should become familiar with the information supplied in order to be able to use it when the structure of a 68000 program does require it. The specific areas to be discussed are stack pointer initialization, multiple module programs, heap initialization for use with the dynamic memory routines (NEW, DISPOSE, MARK, and RELEASE) and interrupt processing with Pascal programs.

REGISTER ALLOCATION

The 68000 has eight data (D) registers and eight address (A) registers. The 68000 makes use of all 16 registers. Addresses are computed only in the A registers while data values are computed only in the D registers. Data registers 0 through 6 are used for expression evaluations. Data register 7 is the function value return register. In other words, if the value of a function is 4 bytes or less, the value will be returned in D7. Address registers 5, 6, and 7 are permanently allocated by the compiler for specific tasks. The user must never destroy the addresses contained in these registers.

A7 is the stack pointer; it always points to the last item on the stack. A6 is the local frame pointer. It always points to the highest address of the data area of the currently executing procedure or function.

The LINK and UNLK instructions are used by the compiler with A6 and A7 to maintain a linked list of local data areas for nested procedure calls.

A5 is the static data pointer. If the $COMMON$ option is used to access variables anywhere in the program, then A5 must be initialized to the same value that was specified for A5 at link time. If the $COMMON$ option is never used (remember that $COMMON$ is ON by default), then register A5 will not be used by the compiler. If the $COMMON$ is used for accessing program-level variables, use the library A5_LIB:L68000 when linking. If either $BASE_PAGE$ or $FAR$ is used for accessing global variables, use the library ABS_LIB:L68000 when linking.

STACK POINTER INITIALIZATION

The stack pointer (Address Register 7) is a hardware register maintained by the processor. Prior to use, however, it must be initialized by the user. A program that has a main code section will generate the following statements in the relocatable file:

```
PROG_name  LEA      Prog_Name+8[PC],A0
           JMP      Zstartprogram[PC]
            .
            .
            .  Pascal code
            .
            .
           JMP      Zendprogram [PC]
```

The purpose of Zendprogram is to return control to the user's operating system upon completion. Following is an example of how Zendprogram might be written for use in the emulator:

```
"68000"
               GLOBAL Zendprogram
               EXTERNAL MONITOR_MESSAGE, JSR_ENTRY
Zendprogram
               MOVE #MESSAGE,MONITOR_MESSAGE
               JSR JSR_ENTRY
               JMP $
               DATA
MESSAGE        DC.B END_MESSAGE-MESSAGE-1
               ASC "End of program "
END_MESSAGE
```

This will cause the message "End of program" to appear on the status line if the program runs to completion in the emulator. A Zendprogram like the above example is provided in the run time libraries for use in emulation.

The purpose of Zstartprogram is to allow the user to set up the stack pointer and any other registers before executing the program. An example of a Zstartprogram that can be used to emulate a small program is shown below. For this example, we assume, when linking, that the value of DATA will be 4000H and that the value of A5 will be 0000H. Therefore, A5 must be cleared to zero so that references of the form d[A5] will operate correctly. The static data area will begin at 4000H and go up. The stack will begin at 4000H (predecrement addressing) and go down.

```
"68000"
                  GLOBAL Zstartprogram

Zstartprogram     MOVE.L   #0,A5
                  MOVE.L   #0,A6
                  MOVEA.L  #4000H,A7
                  JMP      [A0]
```

1. A5 and A6 are cleared.
2. The stack pointer is set to 4000H.
3. Jump to the return address in A0.

Another approach to stack pointer initialization is to initialize the stack pointer and address registers 5 and 6 at load time; i.e., the user's operating system would initialize everything prior to entering the program. In this case, Zstartprogram would consist of the instruction JMP [A0] only. Still another way to initialize the stack is to declare the stack as a data structure in the main program as in the following example:

```
(file MODULE1:source)
        .
        .
        .
     VAR
        ...;
        ...;
        $GLOBVAR ON$
        $ORG 10000H
        STACK_AREA : ARRAY[0...7FFFH] of SIGNED_16
        STACK_ : SIGNED_16,
        $END_ORG$
        $GLOBVAR OFF$


     BEGIN
       ...;
       ...;
       ...;
     END.
```

For this case, Zstartprogram might be:

```
MOVE  #STACK_,A7
MOVE  #0,A5
MOVE  #0,A6
JMP   [A0]
EXTERNAL STACK_
```

This would cause the address of STACK_ (20000H in this example) to be loaded into the stack pointer.

NOTE

> STACK_AREA must start on an even address and must be of even length. Defining STACK_AREA in terms of SIGNED_16's guarantees STACK_ to be on an even word boundary.

The use of an absolute address for the stack in the above example gives the user the convenience of assigning a fixed block of memory for the stack. It may be better, however, to allow the compiler to actually preserve a relocatable data area for the stack by leaving out the ORG and END_ORG options. This will help prevent accidental reuse of the assigned stack area by another module.

The following version of Zstartprogram is included in the run time libraries A5_LIB and ABS_LIB. It allocates a default stack of 256 words in the data area:

```
"68000"
                    GLOBAL Zstartprogram
Zstartprogram
                    MOVEA.L    #0,A5
                    MOVEA.L    #0,A6
                    LEA        Zstack,A7
                    JMP        [A0]
                    DATA
                    DS.W       256
Zstack              DS.W       1
                    END
```

DATA TYPE CONSIDERATIONS

Since the 68000 supports 32-bit integers, the default size of integer is 32 bits. If 16 bits are sufficient, however, significantly more efficient code can be generated for multiplication and division. Thus, it is recommended that 16-bit integers be used whenever possible. If only some variables are to be 16 bits, they should be declared individually with:

```
VAR X:SIGNED_16;
```

If all integers are to be 16 bits, this can be done in the type section as in:

```
TYPE INTEGER = SIGNED_16;
```

Using this type definition will cause all subsequent references to INTEGER to be treated as 16-bit signed integers instead of 32.

Pointers are allocated 4 bytes of memory.

The value of false is a byte of all zeros. True is the value 1.

PROCEDURES AND PARAMETERS PASSING

All parameters are passed on the stack. A reference parameter is passed by pushing its address on the stack. A value parameter of 4 bytes or less is passed by pushing the value on the stack. Value parameters greater than 4 bytes have their addresses pushed on the stack like reference parameters. Then at the procedure entry, the large value parameters are copied into the local data area of the procedure. The local data of the procedure is also allocated on the stack at the procedure entry point.

When calling a procedure, the parameters are first pushed on the stack. If the parameter is a byte, it is placed in the most significant byte of the word on the stack with the least significant byte untouched. Parameters that are longer than one word, are pushed one word at a time, with the word which is uppermost in memory pushed first.

Parameters that are passed by reference (VAR parameters) have their addresses pushed on the stack. Note that for large arrays, it is more efficient to use a VAR parameter than a value parameter. The address is pushed in both cases, but the parameter will be copied in the value case. After all the parameters are pushed, the routine is called. Upon return from the routine, the compiler causes the stack pointer to be incremented in order to remove the parameters.

STATIC LINKS

Inner level procedures must have access to the data and parameters of the outer level procedures containing them. In order to do this, static links are used. Whenever a procedure of level 2 or greater is called, a static link is pushed on the stack. This value is a pointer to the static link of the procedure containing the procedure being called. Since level 1 procedures do not have static links, the static link of the level 2 procedure points to where the static link would be. Static links are pushed on before any parameters.

FUNCTIONS

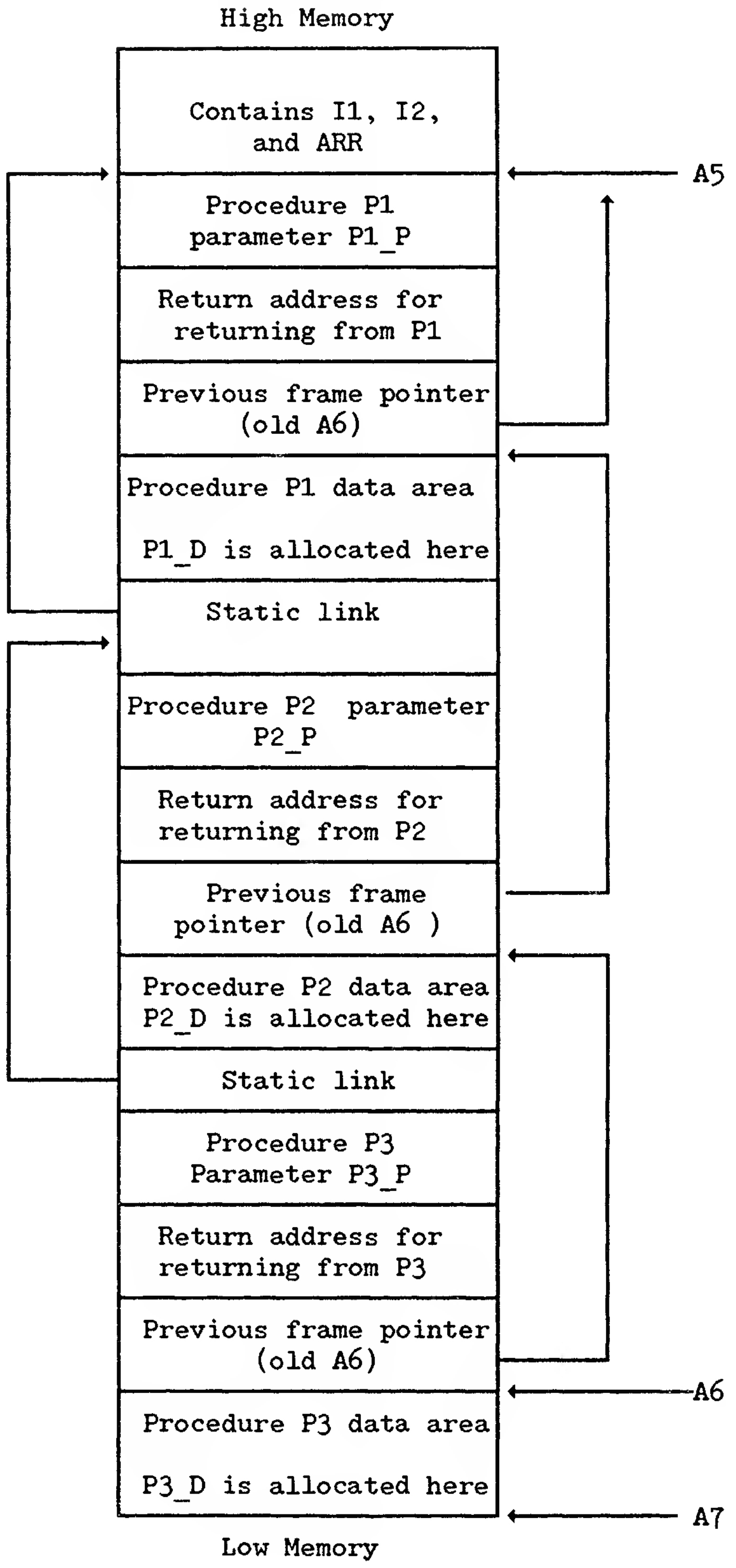
Functions are treated like procedures except that they return a value. If the value fits in four bytes, it will be returned in data register 7.

If the value returned is larger than four bytes, the calling procedure passes an address for the return value. This address is passed after the last parameter immediately before the procedure is called. When incrementing the stack after the call to remove the parameters, the function value return address is also removed.

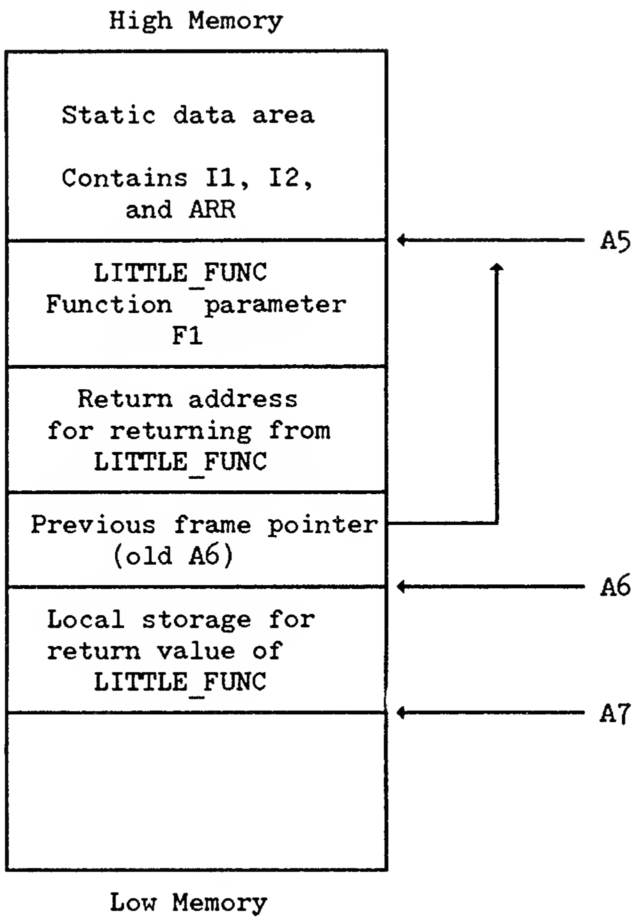
The following sample program illustrates some principles of stack organization:

```
"68000"
PROGRAM TEST;



    TYPE
        BIG_ARRAY = ARRAY[0..10] OF INTEGER;
    VAR
        I1,I2:INTEGER;
        ARR:BIG_ARRAY;

    PROCEDURE P1(P1_P:INTEGER);
        VAR P1_D:INTEGER;

        PROCEDURE P2(P2_P:INTEGER);
            VAR P2_D:INTEGER;

            PROCEDURE P3(P3_P:INTEGER);
                VAR P3_D:INTEGER;

                BEGIN  (P3)
                    P3_P: = 5;
                END;

            BEGIN  (P2)
                P3(P2_P);
            END;

        BEGIN (P1)
            P2(P1_P);
        END;

    FUNCTION LITTLE_FUNC(F1:INTEGER):INTEGER;
        BEGIN
            LITTLE_FUNC :=F1;
        END;

    FUNCTION BIG_FUNC(F1,F2:INTEGER):BIG_ARRAY;
        VAR ARR1:BIG_ARRAY;
        BEGIN
            ARR1[F1] := F2;
            BIG_FUNC := ARR1;
        END;

BEGIN   (TEST)
    P1(I1);
    I1 := LITTLE_FUNC(I2);
    ARR := BIG_FUNC(I1,I2);
END.
```

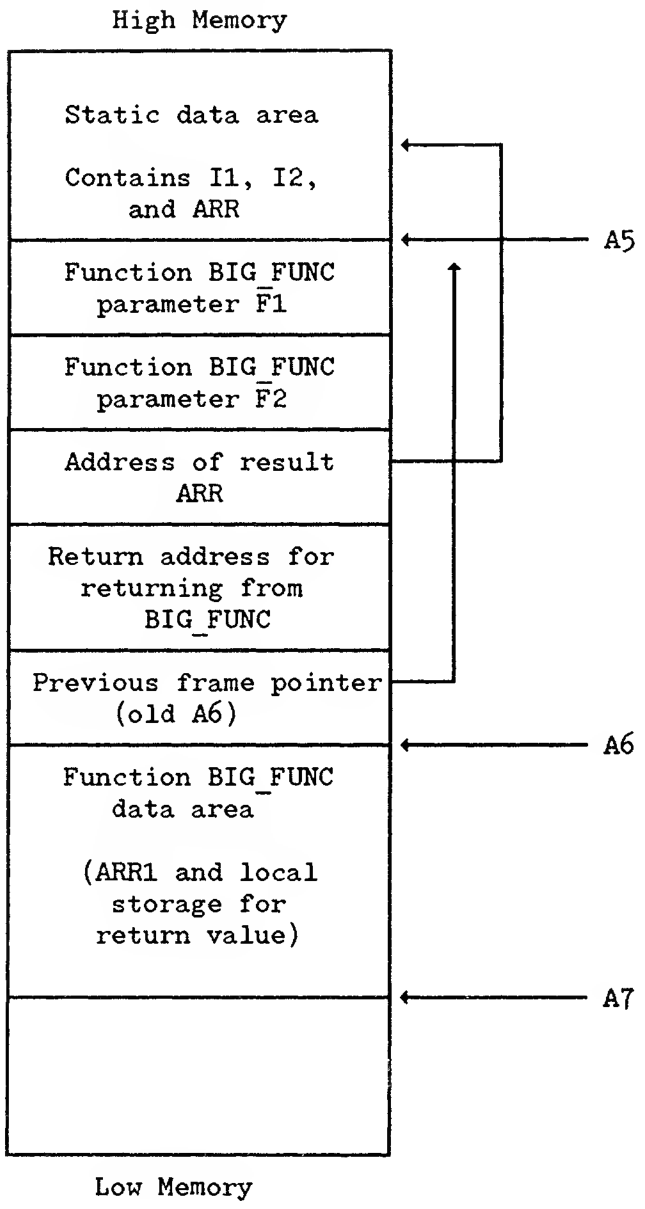
When executing line 22, the stack appears as follows:


High Memory
Contains I1, I2, and ARR
Procedure P1 parameter P1_P
Return address for returning from P1
Previous frame pointer (old A6)
Procedure P1 data area
P1_D is allocated here
Static link
Procedure P2 parameter P2_P
Return address for returning from P2
Previous frame pointer (old A6)
Procedure P2 data area P2_D is allocated here
Static link
Procedure P3 Parameter P3_P
Return address for returning from P3
Previous frame pointer (old A6)
Procedure P3 data area
P3_D is allocated here
A5
A6
A7
Low Memory

When executing line 35, the stack appears as follows:


High Memory
Static data area
Contains I1, I2, and ARR
LITTLE_FUNC Function parameter F1
Return address for returning from LITTLE_FUNC
Previous frame pointer (old A6)
Local storage for return value of LITTLE_FUNC
A5
A6
A7
Low Memory

When executing line 41, the stack appears as follows:


High Memory
Static data area
Contains I1, I2, and ARR
A5
Function BIG_FUNC parameter F1
Function BIG_FUNC parameter F2
Address of result ARR
Return address for returning from BIG_FUNC
Previous frame pointer (old A6)
A6
Function BIG_FUNC data area
(ARR1 and local storage for return value)
A7
Low Memory

Note that for BIG_FUNC, the address of the result (ARR) is passed, since it is larger than four bytes. For LITTLE_FUNC, the address of the result is not passed as the size is only four bytes and the result is returned in D7.

MULTIPLE MODULE PROGRAMS

Only one module in an absolute program file should contain a Pascal program with a main code section. All other modules should contain procedures and functions with a period at the end of the procedure declarations to indicate an empty program block.

Example:

(file MODULE1:source)

```
     PROGRAM MODULE1; {this is the main module}

     CONST
         ...;
     TYPE
         ...;
     VAR
         ...;

     PROCEDURE X(Parameter : Type);EXTERNAL;
     PROCEDURE Y;EXTERNAL;

     BEGIN
         ...;
         ...;              {main code}
         ...;
     END.                  {period signals end of program, main code
                            exists so stack initialization code is
                            generated}
```

NOTE

The transfer address is set to cause execution to begin in the main code section of the program module.

(file MODULE2:source)

```
PROGRAM MODULE2; {this module contains the procedures and
                  functions used in MODULE1}

$GLOBPROC ON$
PROCEDURE X(Parameter : Type);
    BEGIN
      ...;
      ...;
    END;
PROCEDURE Y;
    BEGIN
      ...;
      ...;
    END;
                 {The period signals the compiler that the
                  program has ended.  Since no main code
                  exists, the compiler does not generate any
                  stack initialization code or linker
                  transfer address.}
```

STATIC DATA AREA

The static data area is composed of the static data areas of all the separately compiled Pascal modules. Usually, the static data area is a single continuous block of memory but this need not be the case. It is possible, when linking, to locate the DATA (or PROG) areas of the various modules in several separate areas of memory.

The $COMMON$, $BASE_PAGE$, and FAR compiler options control the addressing mode used by the 68000 to access the variables in the static data area. Usually, the programmer specifies that all variables in all modules be accessed using the same addressing mode but this need not be the case. It is possible for one module's variables to be accessed with one mode and a different module's variables to accessed with a different mode. It is also possible for a single variable to be using different modes in different modules. See the section on Program-level variable addressing in Chapter 1 for more information.

DYNAMIC ALLOCATION HEAP INITIALIZATION

Before using the standard procedures NEW and MARK, the block of memory that you wish to have managed as a dynamic memory allocation pool (the heap) must be initialized by calling the external library procedure:

```
                  INITHEAP(Start_address:Pointer_type;
                Length_in_bytes:UNSIGNED_32); EXTERNAL;
```

The procedure INITHEAP must be declared external as above. Pointer_type is a Pascal pointer. Start_address should point to the smallest address of the memory block to be used, and it must be an even address.

Length_in_bytes must be an even value.

For example, if the block to be used is located in memory from 4000H to 5FFFH, the initialization should appear as follows:

```
     PROGRAM Test;

     CONST
        Heapsize = UNSIGNED_32(2000H);
             .
     TYPE
        Pointer_type = ^INTEGER;
             .
     VAR
     $ORG = 4000H$
        Heapstart:INTEGER;
     $END_ORG$
             .
     PROCEDURE
     INITHEAP(Start_address:Pointer_type;
     Length_in_bytes:UNSIGNED_32); EXTERNAL;
             .
             .
     BEGIN  {main program block}
        INITHEAP(ADDR(Heapstart),Heapsize);
             .
             .
             .
     END.
```

Twelve bytes of heap space are used when the heap space is initialized. Twelve bytes of heap space are also used each time the heap is marked. When NEW is called, the minimum memory allocated is 8 bytes, even if fewer bytes are required.

Items must be allocated in an even number of bytes. The compiler ensures that NEW and DISPOSE always pass even sizes. If the user utilizes these procedures on his own, he must always pass even sizes. Likewise, INITHEAP should always be called with an even address and an even size.

INTERRUPT HANDLING

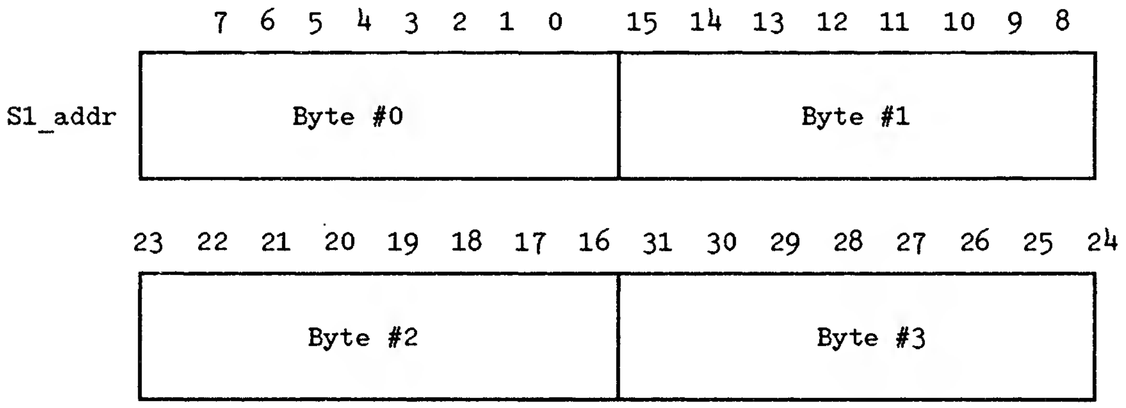
Interrupt handling routines may be written in Pascal using the INTERRUPT option. Additionally, code produced by the compiler is safely interruptable as long as the interrupt driven process saves and restores the registers and returns with a return from exception (RTE) instruction. The compiler does not automatically generate the interrupt vectors for procedures defined as interrupt procedures.

SET SPACE ALLOCATION

The 68000 compiler allocates sets by bytes. The Pascal statements:

```
PROGRAM TEST; VAR S1: SET OF 0..31;
```

will allocate four bytes of data to the set S1. The bits in the set will be numbered as follows:



SPECIAL OPTIONS FOR THE 68000 COMPILER

The following options have special functions for the 68000 compiler.

INTERRUPT
Default OFF
All procedures defined while the INTERRUPT option is on will be suitable for use as interrupt routines. On entry to the procedure, all registers will be saved on the stack. On exit, the registers will be restored and RTE is used to return instead of the normal RTS. INTERRUPT only applies to outer level procedures, as inner level procedures cannot be accessed externally and thus cannot be referenced by an interrupt vector. INTERRUPT procedures may not be called and they may not have parameters. The value of INTERRUPT applies at the PROCEDURE heading. It is the user's responsibility to set up the interrupt vectors to point to the appropriate INTERRUPT procedures.

OPTIMIZE
Default OFF
When OPTIMIZE is off, storing indirectly or into an external will cause the contents of certain registers to be forgotten because there is a possibility that their contents will be incorrect. If OPTIMIZE is on, these registers will not be forgotten. It is possible that the code generated will be incorrect, although in most cases it will be correct. An example which produces incorrect code is shown on the following page.

```
FILE:  TEST:L68000      HP Pascal/64000[B.1] 68000 code generator


00000000  1  "68000"
00000000  1  PROGRAM BAD_CODE;
00000000  1  $EXTENSIONS$
00000000  1  LABEL 1;
00000000  1  VAR I:INTEGER;
00000004  1      J,K:INTEGER;
0000000C  1      L:INTEGER;
00000010  1      IP:^INTEGER;
00000014  1  BEGIN
    00000000     BAD_CODE
    00000000         JSR      Zstartprogram[PC]
    00000004         LINK     A6,#0
00000008  1     IP := ADDR(I);
    00000008         LEA      BAD_COD00_D[A5],A0
    0000000C         MOVE,L   A0,BAD_COD00_D+00010H[A5]
00000010  1     J :=3;
    00000010         MOVE.L   #3_BAD_COD00_D+00004H[A5]
00000018  1     K :=2;
    00000018         MOVE.L   #2,BAD_COD00_D+00008H[A5]
00000020  1  1:
00000020  1     I := J+K;
    00000020     BAD_COD00_1
    00000020         MOVE.L   BAD_COD00_D+00004H[A5],D0
    00000024         ADD.L    BAD_COD00_D+00004H[A5],D0
    00000028         MOVE.L   D0,BAD_COD00_D[A5]
0000002C  1     IP^ :=K;
    0000002C         MOVEA.L  BAD_COD00_D+00010H[A5],A0
    00000030         MOVE.L   BAD_COD00_D+00008H[A5],[A0]
00000034  1     L := I;
    00000034         MOVE.L   D0,BAD_COD00_D+0000CH[A5]
00000038  1  END.
    0000003  JMP     Zendprogram[PC]
    00000000         DATA
    00000000     BAD_COD00_D
    00000000         DS.B     00014H
                     GLOBAL   BAD_CODE
                     EXTERNAL Zstartprogram
                     EXTERNAL Zendprogram
                     END.     BAD_CODE

End of compilation, number of errors=     0
```

If OPTIMIZE is on, L will be assigned 5 instead of 2. Errors may also occur if two externals and/or absolutes refer to the same address.

Another optimization occurs when OPTIMIZE is on; forward jumps will be assumed to be within 128 bytes. This affects the IF, WHILE, CASE, FOR, and GOTO statements and saves two bytes for each forward jump. If the label is out of range, an error (number 1200) will be given in pass 3. If this occurs, turn off OPTIMIZE around the line that caused the error. Normally, programs may be compiled with OPTIMIZE on without fear of generating incorrect code.

SEPARATE
Default OFF
SEPARATE has no effect for the 68000. PROG and DATA are always separate.

TRAP
Default -1
When a procedure heading is encountered in the source, the value of TRAP will be checked. If it is a value from 0 to 15, the procedure will be declared as a TRAP procedure. For the option $TRAP=5$ a procedure name will be assigned the trap value 5. When any calls to this procedure are encountered in the source, the compiler will generate a TRAP 5 instruction rather than a JSR instruction. Trap procedures may have parameters like any other procedure. After the value of TRAP has been assigned to a procedure name, the value of TRAP is set to -1. The trap number used where the procedure is declared global must be the same one used where the procedure is declared external. The compiler generates a call to Zentertrap at the entry point of a trap procedure. Trap procedures return via the RTE instruction. It is the user's responsibility to set up the trap vectors to point to the appropriate TRAP procedures.

ADDRESSING OPTIONS

The following three options allow the user to control the addressing mode for any variables defined while the EXTVAR option is on as well as the entire program level data block. One and only one of these options is always ON. The option that is ON when the keyword PROGRAM is by the compiler shall control the addressing mode of all the variables defined in that program. After the keyword PROGRAM, the options only affect externally defined variables.

If the $COMMON$ option is used ($COMMON$ is ON by default), link with library A5_LIB:L68000. Otherwise, link with the library ABS_LIB:L68000.

BASE_PAGE
Default OFF
When the EXTVAR option is on, all external variables defined while $BASE_PAGE$ is on will be accessed by the absolute short addressing mode. The linker will report an error if a base page variable is linked to an address outside the range 0000000H through 0007FFFH or 0FF8000H through 0FFFFFFH. Insert $BASE_PAGE$ before the program heading if absolute short addressing is desired for locally defined program level variables.

FAR
Default OFF
When the EXTVAR option is on, all external variables defined while FAR is on will be accessed by the absolute long addressing mode. Insert FAR before the program heading if absolute long addressing is desired for locally defined program level variables.

COMMON
Default ON
All external variables defined when $COMMON$ is ON will be accessed with the address register indirect plus displacement mode. The address register will be A5. Insert $COMMON$ (or nothing since $COMMON$ is ON by default) before the keyword PROGRAM if A5+d addressing is desired for locally defined program level variables.

The linker will report an error if the calculated displacement is greater than 32767 or less than -32768. The displacement is equal to the actual address of the referenced variable minus the A5 value that was specified to the linker.

The following four options allow the user to control the addressing modes for calling procedures and functions. One and only one of these options is always on. The addressing mode used to call a procedure or function is determined by the setting of the options when the procedure or function heading is encountered.

CALL_ABS_LONG
Default OFF
Procedures and functions defined while $CALL_ABS_LONG$ is on will be called with the absolute long addressing mode.

CALL_ABS_SHORT
Default OFF
Procedures and functions defined while $CALL_ABS_SHORT$ is on will be called with the absolute short addressing mode. The linker will report an error if an attempt is made to call a procedure or function with the absolute short addressing mode if the procedure or function is not assigned an address outside the range 0000000H through 0007FFFH or 0FF8000H through 0FFFFFFH.

CALL_PC_SHORT
Default ON
Procedures and functions defined while $CALL_PC_SHORT$ is on will be called using the program counter plus displacement addressing mode. The linker will report an error if an attempt is made to call such a procedure or function whose displacement from the current program counter is greater than +- 32K bytes.

CALL_PC_LONG
Default OFF
Procedures and functions defined while $CALL_PC_LONG$ is on will be called using the program counter plus displacement + index addressing mode. This option should be used only when necessary because loading the index portion for each call is inefficient.

The following four options allow the user to control the addressing modes used for calling predefined procedures and functions (i.e., NEW, DISPOSE, etc.). They are similar to the previous four options, but they are applied to each procedure or function call individually. One and only one of these options is always on.

LIB_ABS_LONG
Default OFF
Calls to predefined procedures and functions encountered while LIB_ABS_LONG is on will use the absolute long addressing mode.

LIB_ABS_SHORT
Default OFF
Calls to predefined procedures and functions encountered while LIB_ABS_SHORT is on will use the absolute short addressing mode. The linker will report an error if an attempt is made to call a predefined procedure or function with the short absolute mode and the address of the procedure or function is not in the range 0000000H through 0007FFFH or 0FF8000H through 0FFFFFFH.

LIB_PC_SHORT
Default ON
Calls to predefined procedures and functions encountered while LIB_PC_SHORT is on will use the program counter plus displacement addressing mode. The linker will report an error if an attempt is made to call such a procedure or function and the displacement from the call is greater than +-32K bytes.

LIB_PC_LONG
Default OFF
Calls to predefined procedures and functions encountered while LIB_PC_LONG is on will use the program counter plus displacement plus index addressing mode. This option should be used only when necessary, because loading the index for each call is inefficient.

USER-DEFINED OPERATORS

GENERAL

Pascal/64000 allows the user to redefine the meaning of certain operators. User defined operators are created by using the option: $USER_DEFINED$ during the declaration of a user type. The option, when used, applies to the next type definition encountered.

For user defined operators, the compiler will not generate in-line code to perform the operations; instead, it will generate calls to user provided run-time routines. The run-time routine names will be a composite of the user's type name and the operation being performed: TYPENAME_OPERATION. The first eleven characters of the user's type name are concatenated with an underscore and three characters identifying the operation.

OPERATIONS THAT MAY BE REDEFINED

The following is a list of operators that can be redefined associated with the routine that the compiler will create for the operation.

Operation	Symbol	Run-time Routine
1. Add	+	<typename>_ADD
2. Negate	-	<typename>_NEG
3. Subtract	-	<typename>_SUB
4. Multiply	*	<typename>_MUL
5. Divide	/ or DIV	<typename>_DIV
6. Modulus	MOD	<typename>_MOD
7. Equal Comparison	=	<typename>_EQU
8. Not Equal Comparison	<>	<typename>_NEQ
9. Less Than or Equal to Comparison	<=	<typename>_LEQ
10. Greater Than or Equal to Comparison	>=	<typename>_GEQ
11. Less Than Comparison	<	<typename>_LES
12. Greater Than Comparison	>	<typename>_GTR

The compiler will provide the user with a Store routine. The 68000 compiler will use a multi-byte move loop for types larger than four bytes, or a regular move for types smaller than four bytes.

PARAMETERS FOR USER DEFINED OPERATIONS

For the 68000, the parameters are passed on the stack as follows:

1) The address of the first operand is pushed on the stack.

2) The address of the second operand is pushed on the stack.

3) The address of the result is pushed on the stack if the result is larger than 4 bytes. Otherwise, the compiler expects the result to be returned in data register 7.

Negate has only one operand and a result.

Relational operations will not pass an address for the result. Instead, a Boolean value should be returned in data register 7 as follows:

True:	D7 set to 1
False:	D7 set to 0

User routines may be written in Pascal. For example:

```
FUNCTION REAL_MUL (VAR OPERAND1,OPERAND2:REAL):REAL; (where the
        type REAL has been defined by the user)

FUNCTION REAL_LES (VAR OPERAND1,OPERAND2:REAL):BOOLEAN;
```

NOTE

All parameters are passed by reference (VAR parameters). Functions with result values smaller than 5 bytes return the result value in D7.

The following example is an expanded listing demonstrating use of a user type "REAL".

```
FILE: USER_T:ZZ       HP Pascal/64000[B.1] 68000 code generator


 1 00000000  1  "68000"
                        EXTERNAL RE1
                        EXTERNAL RE2
                        EXTERNAL RE3
                        EXTERNAL SEMAPHORE
 2 00000000  1  $EXTENSIONS+$
 3 00000000  1  PROGRAM USER_TYPE_DEMO;
 4 00000000  1   TYPE
 5 00000000  1    $USER_DEFINED$
 6 00000000  1    REAL=
 7 00000000  1     RECORD
 8 00000000  2      MANTISSA:ARRAY[0..2] OF BYTE;
 9 00000000  2      EXPONENT:BYTE;
10 00000000  2     END;
11 00000000  1
12 00000000  1    VAR
13 00000000  1     $EXTVAR+$
14 00000000  1     RE1,RE2,RE3:REAL;
15 00000000  1     SEMAPHORE:BOOLEAN;
16 00000000  1    BEGIN
      00000000      USER_TYPE_DEMO
      00000000          JSR      Zstartprogram[PC]
      00000004          LINK     A6,#-4
17 00000008  1     RE1:=RE2 - RE3 * RE1;
      00000008          PEA      RE3
      0000000C          PEA      RE1
      00000010          JSR      REAL_MUL[PC]
      00000014          ADDQ.L   #8,A7
      00000016          PEA      RE2
      0000001A          MOVE.L   D7,-4[A6]
      0000001E          PEA      -4[A6]
      00000022          JSR      REAL_SUB[PC]
      00000026          ADDQ.L   #8,A7
      00000028          MOVE.L   D7,RE1
18 0000002C  1     IF -RE1 < RE2 THEN
      0000002C          PEA      RE1
      00000030          JSR      REAL_NEG[PC]
      00000034          ADDQ.L   #4,A7
      00000036          MOVE.L   D7,-4[A6]
      0000003A          PEA      -4[A6]
      0000003E          PEA      RE2
      00000042          JSR      REAL_LES[PC]
      00000046          ADDQ.L   #8,A7
      00000048          TST.B    D7
      0000004A          BEQ      USER_TY00_L1
19 0000004E  1      RE1 :=RE2;
      0000004E          MOVE.L   RE2,RE1
      00000054      USER_TY00_L1
20 00000054  1     SEMAPHORE := RE1 <= RE2;
      00000054          PEA      RE1
      00000058          PEA      RE2
      0000005C          JSR      REAL_LEQ[PC]
      00000060          ADDQ.L   #8,A7
      00000062          MOVE.B   D7,SEMAPHORE
21 00000066  1     END.
                        UNLK     A6
      00000066          JMP      Zendprogram[PC]
      00000000          DATA
      00000000      USER_TY00_D
      00000000          DS.B     00000H
                        GLOBAL   USER_TYPE_DEMO
                        EXTERNAL REAL_NEG
                        EXTERNAL REAL_SUB
                        EXTERNAL REAL_MUL
                        EXTERNAL REAL_LEQ
                        EXTERNAL REAL LES
                        EXTERNAL Zstartprogram
                        EXTERNAL Zendprogram
                        END      USER_TYPE_DEMO

End of compilation, number of errors=  0
```

PASS 2 ERRORS

Pass 2 errors will be displayed on the screen with the message:

```
        LINE # <line number>--PASS2 ERROR # <Pass2 error number>
```

In addition, if a listing file has been indicated for the compilation, the compiler will indicate pass 2 errors where they occurred in the listing. It will also list the meaning of each error.

Pass 2 error numbers will always be >=1000. Errors with numbers between 1000 and 1099 are fatal errors. Errors with numbers >=1100 are non-fatal errors.

Pass 2 will stop generating code after a fatal pass 2 error. If a listing file has been indicated for the compilation, pass 3 will give you a listing with errors. Non-fatal errors are output to the display and to the listing file (if one exists), but compilation continues after appropriate action has been taken to correct the error. A list of pass 2 errors is given in Table 2-1.

Table 2-1. 68000 Pass 2 Errors

1000 - "Out of memory"
The 68000 code generator has run out of memory, break up your program and recompile. This error can also occur if there is a bug in the compiler itself. If you feel that you have not used up the entire symbol table, contact Hewlett-Packard.

1001 - "This error can't possibly occur #1"
Contact Hewlett-Packard.

1002 - "Size error"
A size larger than the maximum size allowed for a type has been detected.

1003 - "This error can't possibly occur #2"
Contact Hewlett-Packard.

1004 - "Type error"
An operation with an incorrect type of operand has been detected; for example, a negation of an unsigned value.

1005 - "Unimplemented feature"
You have used a feature of Pascal that has not been implemented in the 68000 code generator.

1006 - "Compiler error. Contact Hewlett-Packard"
This error should never occur. Please report this error to Hewlett-Packard as soon as possible.

1008 - "All data registers are active"
Even though the 68000 has 8 data registers, it is still possible to write an expression that will require more. Break up the expression and recompile.

1009 - "All address registers are active"
The compiler computes addresses in address registers A0 - A4. You have succeeded in writing an expression that requires computation of more than 5 addresses. Break up the expression and recompile.

1100 - "Bounds error"
An attempt was made to store a value into a result which was too small; for example assigning 300 to a byte. This error will occur if the $RANGE$ option is on.

1103 - "Interrupt procedure must not have parameters"
An interrupt procedure can not have parameters.

1104 - "Interrupt procedure call not allowed"
An interrupt routine can only be accessed through an interrupt vector, since it will return with an RTE instead of an RTS.

1105 - "Data size too large"
More than 32K bytes of data have been allocated for this procedure.

1106 - "Trap or interrupt routine may not be a function"
Only procedures may be trap or interrupt routines.

1107 - "Data counter overflow"
The DATA section has become larger than 32K bytes.

1108 - "Trap number must be 0 to 15"
The 68000 has 16 trap vectors numbered 0 to 15.

1113 - "Program counters do not agree"
If this error occurs before any other error then it means there is a bug in the compiler - contact Hewlett-Packard. If this error occurs with some other error, ignore it.

1200 - "Long range error; turn off OPTIMIZE for this line"
The compiler has tried to generate an 8-bit jump where a 16-bit jump is required. Turn off the OPTIMIZE option around the source line where the error is reported, and recompile.

Chapter 3

RUN-TIME LIBRARY SPECIFICATIONS

INTRODUCTION

This chapter describes the run-time library routines needed to execute programs compiled by the Pascal/64000 compiler for the 68000/68010 and 68008 microprocessors. There are seven libraries provided for the 68000/68010 and seven libraries for the 68008 processor. The seven libraries for the 68000/68010 are identical to the libraries for the 68008 except tor the processor name used when the libraries were produced. All library files for the 68000/68010 processor are in userid L68000, while library files for the 68008 processor are in userid L68008.

Pascal I/O functions are provided by the libraries A5_PIOLIB:L68000, ABSPIOLIB:L68000, A5_PIOLIB;L68008, and ABSPIOLIB:L68008. A complete description of the library routines is contained in the Pascal/64000 Reference Manual. A5_PIOLIB and ABSPIOLIB define the same routines and are identical in function except that A5_PIOLIB uses A5 + d addressing to reference global data while ABSPIOLIB uses absolute addressing.

Simulated I/O functions for the emulation environment are provided by A5_SIMLIB:L68000, ABSSIMLIB:L68000, A5_SIMLIB:L68008, and ABSSIMLIB:L68008. A complete description of the library routines is contained in the Pascal/64000 Reference Manual. A5_SIMLIB and ABSSIMLIB define the same routines and are identical in function except that A5_SIMLIB uses A5 + d addressing to reference global data while ABSSIMLIB uses absolute addressing.

Real number operations are provided by REAL_LIB:L68000 and REAL_LIB:L68008. The operation of these routines is described later in this chapter.

Dynamic memory allocation, 32-bit arithmetic, string, and other operations are provided by A5_LIB:L68000, ABS_LIB:L68000, and ABS_LIB:L68008. The operation of these routines is described later in this chapter. A5_LIB and ABS_LIB define the same routines and are identical in function except that A5_lib uses A5 + d addressing to reference global data while ABS_LIB uses absolute addressing.

Usually, programmers use one access method, COMMON, BASE_PAGE, or FAR in all separately compiled modules of a program. In this case, if use $COMMON$ ($COMMON$ is ON by default) for accessing program-level variables, link to the "A5_" versions of the various libraries. Otherwise, if your program uses $BASE_PAGE$ or $FAR$, link to the "ABS" versions of the libraries. If you mix accessing modes, you can use either library. If you use the "A5_" versions, you must insure that the run-time value of A5 is equal to the value of A5 specified at link time.

DYNAMIC MEMORY ALLOCATION

Pascal/64000 supports dynamic allocation and deallocation of storage space through the procedures NEW, DISPOSE, MARK, RELEASE, INITHEAP, and INCREASEHEAP.

INITHEAP

The user declares a block of memory to be used as the memory pool or heap by calling: INITHEAP (Start_address:Pointer_type; Length_in_bytes:UNSIGNED_32). The procedure, INITHEAP, must be declared EXTERNAL in the declaration block of a program. The resultant heap will be 12 bytes smaller than length_in_bytes.

NEW

The procedure NEW (Pointer : Pointer_to_type) is used to allocate space. The procedure, NEW, searches for available space in a free-list of ascending size blocks. When a block is found that is the proper size or larger, it is allocated and any space left over is returned to the free-list in a new place corresponding to the size of the leftover block. A minimum of 8 bytes is allocated even when fewer than 8 bytes are required.

DISPOSE

The procedure DISPOSE (Pointer : Pointer_to_type) is exactly the reverse of the procedure NEW. It indicates that storage occupied by the indicated variable is no longer required.

MARK

The procedure MARK (Pointer : Pointer_to_type) marks the state of the heap in the designated variable that may be of any pointer type. The variable must not be subsequently altered by assignment.

RELEASE

The procedure RELEASE (Pointer : Pointer_to_type) restores the state of the heap to the value in the indicated variable. This will have the effect of disposing all heap objects created by the NEW procedure since the variable was marked. The variable must contain a value returned by a previous call to MARK; this value may not have been passed previously as a parameter to RELEASE.

32-BIT UNSIGNED ARITHMETIC

Zunsmult - Unsigned 32-bit multiply

FUNCTION Zunsmult(P1,P2: UNSIGNED 32):UNSIGNED_32; Multiplies the unsigned 32-bit number in P1 by the unsigned 32-bit number in P2. The 32-bit result is returned in D7. Dunsmult is the same as Zunsmult except that a TRAPV instruction will be executed in Dunsmult if an overflow occurs. Dunsmult will be called instead of Zunsmult if the DEBUG option is on when a multiply is encountered. Zunsmult and Dunsmult pop the parameters off the stack before returning.

Zmult - Signed 32-bit multiply

FUNCTION Zmult(P1,P2: UNSIGNED_32): UNSIGNED_32; Multiplies the 32-bit number in P1 by the signed 32-bit number in P2. The signed 32-bit result is returned in D7. Dmult is the same as Zmult except that a TRAPV instruction will be executed in Dmult if an overflow occurs. Dmult will be called instead of Zmult if the DEBUG option is on when a multiply is encountered. Zmult and Dmult pop the parameters off the stack before returning.

Zunsdiv - Unsigned 32-bit divide

FUNCTION Zunsdiv(P1,P2:UNSIGNED_32):UNSIGNED_32; Divides the unsigned 32-bit number in P1 by the unsigned 32-bit number in P2. The 32-bit result is returned in D7. A zero divide exception will be executed if division by zero is attempted. Zunsdiv pops the parameters off the stack before returning.

Zdiv - Signed 32-bit divide

FUNCTION Zdiv(P1,P2:SIGNED_32):SIGNED_32; Divides the signed 32-bit number in P1 by the signed 32-bit number in P2. The signed 32-bit result is returned in D7. A zero divide exception will be executed if division by zero is attempted. Zdiv pops the parameters off the stack before returning.

Zmods - Signed 32-bit modulus

FUNCTION Zmods(P1,P2:SIGNED_32): SIGNED_32; Computes P1 MOD P2 using Zdiv. The 32-bit signed modulus is returned in D7. Zmods pops the parameters off the stack before returning.

Zmodu - Unsigned 32-bit modulus

FUNCTION Zmodu(P1,P2:UNSIGNED_32):UNSIGNED_32; Computes P1 MOD P2 using Zunsdiv. The 32-bit unsigned modulus is returned in D7. Zmodu pops the parameters off the stack before returning.

STRING OPERATIONS

Zstr_comp - String compare

PROCEDURE Zstr_comp(VAR P1,P2:STRING); Compares the strings P1 and P2. This procedure sets the 68000 condition codes to indicate the result of the comparison. Zstr_comp pops the parameters off the stack before returning.

Zchar_str_comp - Character to string compare

PROCEDURE Zchar_str_comp(VAR P1:string; P2:char); Compares the character P2 to the string P1. The character is treated as a string of length 1, and the comparison made as in Zstr_comp, although Zstr_comp is not used. Zchar_str_comp pops the parameters off the stack before returning.

Zstr_move - String move

PROCEDURE Zstr_move(VAR FROM,TO:STRING); This procedure is called to move a string from one address to another. The run time length of the string is not checked. Zstr_move pops the parameters off the stack before returning.

Dstr_move - String move with length check

PROCEDURE Dstr_move(VAR FROM,TO:STRING;LENGTH:SIGNED_16); When the RANGE option is on, this procedure is called in place of Zstr_move to move a string. If the run time length of the string being moved is greater then the length of the block provided, a bounds violation is generated. Dstr_move pops the parameters off the stack before returning.

String equality and inequality in the PASCAL compiler are determined by the following rules:

a. Two strings are equal if their lengths are equal and they are equal character by character.

b. The inequality of two strings is determined at the first character where they differ. If all characters are equal then the longest string is the greater of the two strings.

OTHER LIBRARY ROUTINES

Zenter__trap - Trap procedure entry

A Pascal procedure may be declared to be a TRAP procedure by placing the option $TRAP=n$ immediately before the procedure heading. The value of n is a number in the range 0..15. The compiler will generate a TRAP n instruction rather than calling the procedure whenever the procedure is called in the Pascal source. The procedure may have parameters. If the procedure has parameters, Zenter_trap will be called to copy the parameters on to the system stack. TRAP procedures exit with the RTE instruction.

Zstartprogram - Run time environment initialization

This procedure is called at the beginning of the program block. It provides the user with a place to initialize registers before beginning a program. The user will normally want to provide his own version of Zstartprogram. The version provided clears A5 and A6 and declares a default stack of 256 words.

Zendprogram - Run time environment cleanup

This procedure is jumped to at the end of the program block. Its purpose is to display the message "End of program" on the status line when a program runs to completion during emulation. When a program is executed in an environment where emulation is not used, the user must provide his own version of Zendprogram for the purpose of returning control to his operating system.

Zcase__error - Unspecified case value handler

This procedure is called whenever the computed value of a case expression is not one of the values specified in the case statement. Its purpose is to display the message "Case value at XXXX", where XXXX is the address where the case value error was detected. The error message is displayed on the status line during emulation. When the program is created in an environment where the emulator is not used, the user must provide his own version of Zcase_error.

Dcase__error - DEBUG unspecified case value handler

This procedure is called whenever the computed value of a case expression is not one of the values specified in the case statement and the DEBUG option is on. Its purpose is to display the message "Bad case value in D6 at XXXX", where XXXX is the address where the case value error was detected. The error message is displayed on the status line during emulation. Data register 6 contains the unaltered case value. When the program is executed in an environment where the emulator is not used, the user must provide his own version of Dcase_error.

Zchk - Signed range checking

PROCEDURE Zchk(VALUE,LOW,HIGH:SIGNED_32); This procedure is called to perform range checking of 32-bit signed integers, when the RANGE option is on. If the value being checked is not in range the message "Signed range error at XXXX", where XXXX is the return address for Zchk, is displayed on the status line. Zchk pops the parameters off the stack before returning. When a program is executed in an environment where the emulator is not used, the user must provide his own version of Zchk.

Zunschk - Unsigned range checking

PROCEDURE Zunschk(VALUE,LOW,HIGH:UNSIGNED_32); This proceudre is called to perform range checking of 32-bit unsigned integers, when the RANGE option is on. If the value being checked is not in range the message "Unsigned range error at XXXX", where XXXX is the return address of Zunschk, is displayed on the status line. Zunschk pops the parameters off the stack before returning. When a program, is executed in an environment where the emulator is not used, the user must provide his own version of Zunschk.

NOTE

The declarations in the above procedures are shown only to indicate the calling sequence. Only INITHEAP should be declared external if memory management is used.

FLOATING POINT OPERATION

The compiler handles operations on operands of type REAL or LONGREAL by generating calls to routines contained in the library REAL_LIB:L68000. Although the routines in REAL_LIB are written in assembly language, they can be thought of as Pascal functions, passing parameters and the result according to the conventions described in chapter 2 under Functions. Individual routines or the entire library may be replaced with routines conforming to the following Pascal function descriptions:

Type REAL
Type REAL is a 32-bit, IEEE format, floating point number.

```
Zreal_abs  -  result := ABS(OP);
              FUNCTION Zreal_abs(OP:REAL):REAL;

Zreal_add  -  result := OP1 + OP2;
              FUNCTION Zreal_add(OP1,OP2:REAL):REAL;

Zreal_atan -  result := ARCTAN(OP);
              FUNCTION Zreal_atan(OP:REAL):REAL;

Zreal_cos  -  result := COS(OP);
              FUNCTION Zreal_cos(OP:REAL):REAL;

Zreal_div  -  result := OP1/OP2;
              FUNCTION Zreal_div(OP1,OP2:REAL):REAL;

Zreal_equ  -  result := OP1 = OP2;
              FUNCTION Zreal_equ(OP1,OP2:REAL):REAL;

Zreal_exp  -  result := EXP(OP);
              FUNCTION Zreal_exp(OP:REAL):REAL;

Zreal_geq  -  result := OP1 >= OP2;
              FUNCTION Zreal_geq(OP1,OP2:REAL):REAL;

Zreal_gtr  -  result := OP1 > OP2;
              FUNCTION Zreal_gtr(OP1,OP2:REAL):REAL;

Zreal_leq  -  result := OP1 <= OP2;
              FUNCTION Zreal_leq(OP1,OP2:REAL):REAL;

Zreal_les  -  result := OP1 < OP2;
              FUNCTION Zreal_les(OP1,OP2:REAL):REAL;
```

```
     Zreal_ln    -  result := LN(OP);
                    FUNCTION Zreal_ln(OP:REAL):REAL;

     Zreal_mul   -  result := OP1 * OP2;
                    FUNCTION Zreal_mul(OP1,OP2:REAL):REAL;

     Zreal_neg   -  result := -OP;
                    FUNCTION Zreal_neg(OP:REAL):REAL;

     Zreal_neq   -  result := OP1 <> OP2;
                    FUNCTION Zreal_neq(OP1,OP2:REAL):REAL;

     Zreal_round - result := ROUND(OP);
                    FUNCTION Zreal_round(OP:REAL):REAL;

     Zreal_sin   -  result := SIN(OP);
                    FUNCTION Zreal_sin(OP:REAL):REAL;

     Zreal_sqrt  -  result := SQRT(OP);
                    FUNCTION Zreal_sqrt(OP:REAL):REAL;

     Zreal_sub   -  result := OP1 - OP2;
                    FUNCTION Zreal_sub(OP1,OP2:REAL):REAL;

     Zreal_trunc - result := TRUNC(OP);
                    FUNCTION Zreal_trunc(OP:REAL):REAL;
```

Type LONGREAL
Type LONGREAL is a 64-bit, IEEE format, floating point number.

```
 Zlongreal_abs  -  result := ABS(OP);
             FUNCTION Zlongreal_abs(OP:LONGREAL):LONGREAL;

 Zlongreal_add  -  result := OP1 + OP2;
             FUNCTION Zlongreal_add(OP1,OP2:LONGREAL):LONGREAL;

 Zlongreal_atan -  result := ARCTAN(OP);
             FUNCTION Zlongreal_atan(OP:LONGREAL):LONGREAL;

 Zlongreal_cos  -  result := COS(OP);
             FUNCTION Zlongreal_cos(OP:LONGREAL):LONGREAL;

 Zlongreal_div  -  result := OP1/OP2;
             FUNCTION Zlongreal_div(OP1,OP2:LONGREAL):LONGREAL;
```

```
Zlongreal_equ   -  result := OP1 = OP2;
            FUNCTION Zlongreal_equ(OP1,OP2:LONGREAL):LONGREAL;

Zlongreal_exp   -  result := EXP(OP);
            FUNCTION Zlongreal_exp(OP:LONGREAL):LONGREAL;

Zlongreal_geq   -  result := OP1 >= OP2;
            FUNCTION Zlongreal_geq(OP1,OP2:LONGREAL):LONGREAL;

Zlongreal_gtr   -  result := OP1 > OP2;
            FUNCTION Zlongreal_gtr(OP1,OP2:LONGREAL):LONGREAL;

Zlongreal_leq   -  result := OP1 <= OP2;
            FUNCTION Zlongreal_leq(OP1,OP2:LONGREAL):LONGREAL;

Zlongreal_les   -  result := OP1 < OP2;
            FUNCTION Zlongreal_les(OP1,OP2:LONGREAL):LONGREAL;

Zlongreal_ln    -  result := LN(OP);
            FUNCTION Zlongreal_ln(OP:LONGREAL):LONGREAL;

Zlongreal_mul   -  result := OP1 * OP2;
            FUNCTION Zlongreal_mul(OP1,OP2:LONGREAL):LONGREAL;

Zlongreal_neg   -  result := -OP;
            FUNCTION Zlongreal_neg(OP:LONGREAL):LONGREAL;

Zlongreal_neq   -  result := OP1 <> OP2;
            FUNCTION Zlongreal_neq(OP1,OP2:LONGREAL):LONGREAL;

Zlongreal_round - result := ROUND(OP);
            FUNCTION Zlongreal_round(OP:LONGREAL):LONGREAL;

Zlongreal_sin   -  result := SIN(OP);
            FUNCTION Zlongreal_sin(OP:LONGREAL):LONGREAL;

Zlongreal_sqrt -  result := SQRT(OP);
            FUNCTION Zlongreal_sqrt(OP:LONGREAL):LONGREAL;

Zlongreal_sub   -  result := OP1 - OP2;
            FUNCTION Zlongreal_sub(OP1,OP2:LONGREAL):LONGREAL;

Zlongreal_trunc - result := TRUNC(OP);
            FUNCTION Zlongreal_trunc(OP:LONGREAL):LONGREAL;
```

Appendix A

RUN-TIME ERROR DESCRIPTIONS

When emulating with the monitor linked in, run-time errors will be displayed on the status line. The program will jump to the monitor and the message: 68000---Running ...in monitor eeeeeeeeeeeeeeeeeeee at XXXXXX

where: XXXXXX represents the next address after the call to the

error routine, and eeeeeeeeeeeeeeeeeeeee is an explanation of the error.

Descriptions of possible run-time errors are listed as follows;

Overflow
This error occurs when DEBUG is on. An arithmetic operation caused an overflow.

Bounds error
This error occurs when RANGE is on. When RANGE is on, it will occur if an assignment is made which is outside the bounds specified in the declaration section.

Case error
The case expression evaluated to a value not specified by the case statement, and there was no OTHERWISE statement. If DEBUG was on, the unmodified value will be in D6.

Divide by zero
A division by zero was detected.

Illegal Instruction
The processor attempted to execute an illegal opcode.

I/O error NN
An error occurred in one of the Pascal I/O library routines. The error code NN describes the nature of the error and is defined in the Pascal/64000 Reference Manual.

Memory error #X
An error occurred during dynamic memory allocation (NEW, DISPOSE, MARK, and RELEASE). X indicates errors as follows:

0: Heap length too small (call INITHEAP with larger number for size of heap).

1: Heap has not been initialized (call INITHEAP before first use of NEW or MARK.

2: No free space in current mark. Space may exist in previous marks but is not available to the keyword NEW.

3: No block large enough to allocate but smaller blocks may exist.

4: Pointer variable points outside of heap.

5: No free space in heap.

6: Unable to mark, no block large enough.

7: Attempted to release mark that does not exist.

The address shown with a memory error gives the call to the memory routine, not the call to the error routine. The error number X is passed as a byte parameter to MEMERR.

End of program
Not actually an error; this message is displayed when the program terminates. No address is given. Zendprogram is jumped to instead of being called.

64815-90906, MAY 1984
Replaces: 64815-90905, February 1984

HEWLETT
PACKARD

PRINTED IN U.S.A.